# Algorithmic Complexity Analysis

## Topics

1. Asymptotic Analysis
2. Best, Worst and Average Case Analysis
3. Asymptotic Notations
4. Analysis of Loops
5. Analysis of Recursive Algorithms: Solving Recurrences
6. Amortized Analysis
7. Time Complexity Table
8. Space Complexity
9. Problems & Solutions

## Asymptotic Analysis

**Given two algorithms for a task, how do we find out which one is better?**
One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1. It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better than the first one.
2. It might also be possible that for some inputs, first algorithm performs better on one machine and the second performs better on some other machine for some other inputs.

**Asymptotic Analysis** is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

For example**,** let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slower machine. The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

**Does Asymptotic Analysis always work?**
Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms A and B. A takes $1000 \times n \times \log_2(n)$ and B takes $2 \times n \times \log_2(n)$ time on any given machine. Both of these algorithms are asymptotically same (order of growth is $n \times \log_2(n)$). So, with Asymptotic Analysis, we can't judge which one is better, as we ignore constants in Asymptotic Analysis.

Also, in Asymptotic Analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

## Best, Worst & Average Case Analysis

Consider the following code of Linear Search:

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
        }
        return -1;
}
```

**Worst Case Analysis:**
In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (`key` in the above code) is not present in the array. When `key` is not present, the `linearSearch()` function compares it with all the elements of `arr[]` one by one. Therefore, the worst case time complexity of linear search would be O(*n*).

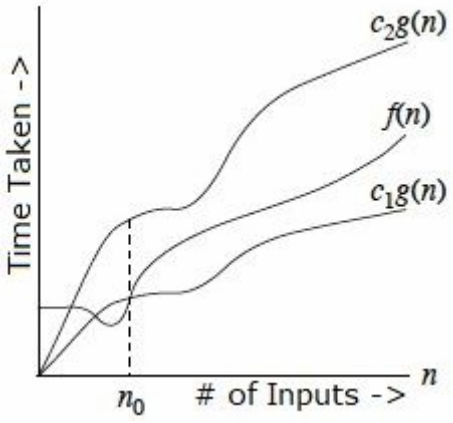
**Best Case Analysis:**
In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when `key` is present at the $0^{th}$ location of `arr[]`, *i.e.*, at `arr[0]`. The number of operations in the best case analysis of Linear Search is constant (not dependent on *n*). So time complexity of Linear Search in the best case would be O(1).

**Average Case Analysis:**
In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of `key` not being present in `arr[]`). [Note: Uniform Distribution is an unbiased distribution, in which all the outcomes are equally probable. Read more about Uniform Distribution **here**]

## Asymptotic Notations

### A. Θ Notation

The Θ (Theta) notation bounds a functions from above and below, so it defines exact asymptotic behavior. A simple way to get Θ notation of an expression is to drop lower order terms and ignore leading constants.




For example, consider the following expression: $3n^3 + 6n^2 + 6000 = \Theta(n^3)$.
Dropping lower order terms is always fine because there will always be a an $n_0$ after which $\Theta(n^3)$ beats $\Theta(n^2)$ irrespective of the constants involved.

For a given function $g(n)$, we denote $\Theta(g(n))$ as the following set of functions:
$\Theta(g(n)) = \{f(n)$: there exist positive constants $c_1$, $c_2$ and $n_0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$.

The above definition means, "if $f(n)$ is theta of $g(n)$, then the value $f(n)$ is always between $c_1 g(n)$ and $c_2 g(n)$ for large values of n ($n \geq n_0$). The definition of theta also requires that $f(n)$ must be non-negative for values of $n$ greater than $n_0$".

**B. Big O Notation**

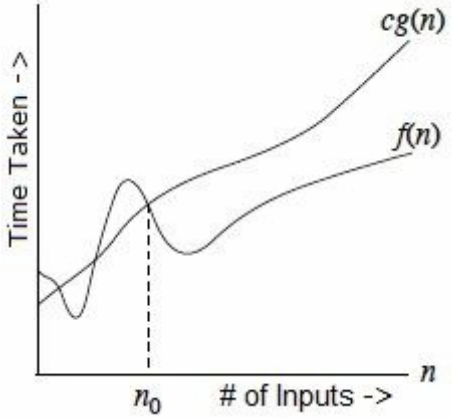
The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time [O($n$)] in best case and quadratic time [O($n^2$)] in worst case. We can safely say that the time complexity of Insertion sort is O($n^2$). Note that O($n^2$) also covers linear time.

If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

- **a.** The worst case time complexity of Insertion Sort is Θ($n^2$).
- **b.** The best case time complexity of Insertion Sort is Θ($n$).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O($g(n)$) = { $f(n)$: there exist positive constants c and $n_0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$}.



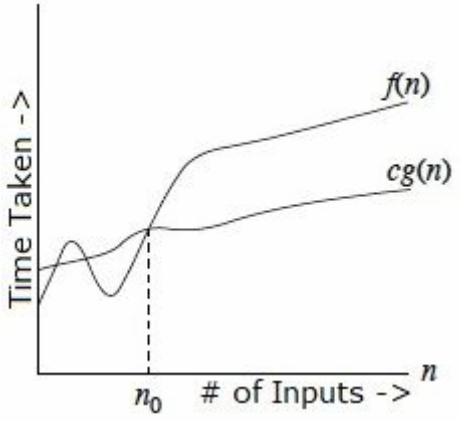

A more mathematically inclined definition of Big O is:

$f(n)$ = O($g(n)$) iff ∃ positive constants $c$ and $n_0$ such that $0 \leq f(n) \leq cg(n)$ ∀ $n \geq n_0$.

### C. Ω Notation

Just as Big O notation provides an asymptotic upper bound on a function, Ω (Omega) notation provides an asymptotic lower bound.
Ω Notation can be useful when we have lower bound on time complexity of an algorithm. The best case performance of an algorithm is generally not useful, hence the Omega notation is the least used notation among all three.

For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions.
$\Omega(g(n)) = \{ f(n)$: there exist positive constants $c$ and $n_0$ such that $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0 \}$. [Note: $f(n)$ and $g(n)$ have swapped places, compared to Big O]




Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as $\Omega(n)$, but it is not a very useful information about insertion sort, as we are generally interested in worst case (Big O) and sometimes in average case (Big Θ).

## Analysis of Loops

1. **O(1):** Time complexity of a function (or set of statements) is considered as O(1), if it doesn't contain any iterative statement(s), recursive statement(s) and call(s) to any other non-constant time function.
   For example, swap() function has O(1) time complexity.

```
void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

   Note that the loop/code that runs a constant number of times is also considered as O(1).

2. **O($n$):** Time Complexity of a loop [iterative statement(s)] is considered as O($n$) if the loop variables is incremented/decremented by a constant amount. For example, following functions have O($n$) time complexity.

```
//Here c is a positive integer constant
```

a.
```
for (int i = 1; i <= n; i += c) {
    // some O(1) expressions
}
```

b.
```
for (int i = n; i > 0; i -= c) {
    // some O(1) expressions
}
```

3. **O($n^c$) :** Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example, the following sample loops have O($n^2$) time complexity.

a.
```
for (int i = 1; i <= n; i += c) {
    for (int j = 1; j <= n; j += c) {
        // some O(1) expressions
    }
}
```

b.
```
for (int i = n; i > 0; i -= c) {
    for (int j = i + 1; j <= n; j += c) {
        // some O(1) expressions
    }
}
```

4. **O($\log_2(n)$) :** Time Complexity of a loop is considered as O($\log_c(n)$) if the loop variables is divided/multiplied by a constant amount.

a.
```
for (int i = n; i > 0; i /= c)
    // some O(1) expressions
```

b.
```
for (int i = 1; i <= n; i *= c)
    // some O(1) expressions
```

For example, Binary Search has O($\log_2(n)$) time complexity.

5. **O($\log_2(\log_2(n))$):** Time Complexity of a loop is considered as O($\log_2(\log_2(n))$) if the loop variables is reduced/increased exponentially by a constant amount.

a.
```
// Here c is a constant, such that c > 1
for (int i = 2; i <= n; i = pow(i, c)) {
    // some O(1) expressions
}
```

b.
```
// Here fun() is sqrt(√) or cuberoot(∛) or any other constant root
for (int i = n; i > 0; i = fun(i)) {
    // some O(1) expressions
}
```

Must Read: **What would cause an algorithm to have O($\log_2(\log_2(n))$) time complexity?**

**How to combine time complexities of consecutive loops?**
When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <= m; i += c) {
     // some O(1) expressions
}
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}
```

- Time complexity of above code is O($m$) + O($n$) which is O($m$ + $n$)
- If $m$ == $n$, the time complexity becomes O(2$n$) which is O($n$).

**How to calculate time complexity when there are many if-else statements inside loops?**
As discussed here, worst case time complexity is the most useful among best, average and worst time complexities. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.

For example consider the linear search function where we consider the case when element is present at the end or not present at all. When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.

## Analysis of Recursive Algorithms: Solving Recurrences

Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size '$n$' as a function of $n$ and the running time on inputs of smaller sizes.

For example, in Merge Sort, to sort a given array, we divide it into two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as T($n$) = 2T($n$/2) + $n$. There are many other recursive algorithms like Binary Search, Towers of Hanoi, Quicksort etc.

There are mainly three ways for solving recurrences relationships:

**(1) Substitution Method:** Recurrence relations themselves are recursive. We solve the recurrence relation by repeated substitution until we get a generic form.

For example, consider the recurrence T(n) = T(n-1) + 1 [Time to solve problem of size n]
where T(0) = 1 [Time to solve problem of size 0 - Base Case]

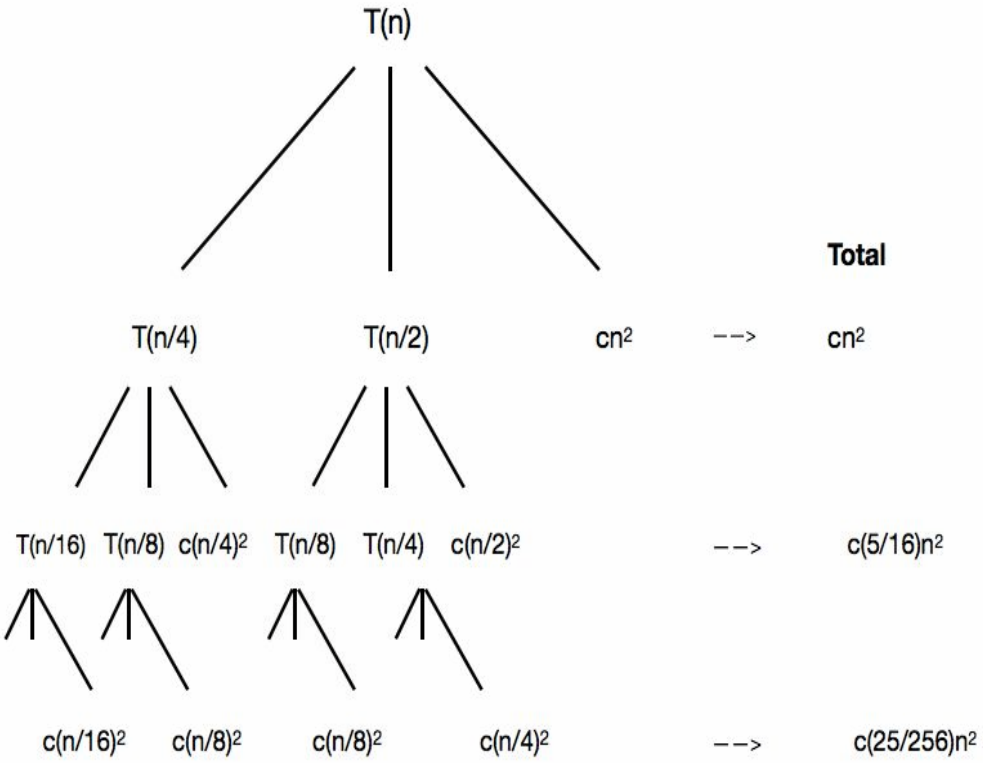
T(n) = T(n-1) + 1
= ( T(n-2) + 1 ) + 1 [T(n-1) = T(n-2) + 1]
= T(n-2) + 2
= ( T(n-3) + 1 ) + 2 [T(n-2) = T(n-3) + 1]
= T(n-3) + 3
= T(n-k) + k

Since we know that: T(0) = 1, therefore we equate n-k to 0, which implies k=n
∴ T(n) = T(n-k) + k = T(n-n) + n = T(0) + n = 1 + n
= O(n) - Linear in Nature

**(2) Recurrence Tree Method:** In this method, we draw a recurrence tree and calculate the total time taken in every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically an Arithmetic Progression [AP] or Geometric Progression [GP].

For example consider the following recurrence relation:

$T(n) = T(n/4) + T(n/2) + cn^2$




To know the value of $T(n)$, we calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series: $T(n) = cn^2(1 + (5/16) + (25/256) + ....)$

The above series is a GP with ratio 5/16. To get an upper bound, we can sum the infinite series. [Sum = $a / (1 - r)$]. We get the sum as $(cn^2)/(1 - (5/16))$, which is $O(n^2)$.

**(3) Master Theorem: [Simplified]**
Master's Theorem is a direct way to get the complexity of a recurrence. It works only for following type of recurrences or for recurrences that can be transformed to following type.

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$
$t = \log_b a$

There are following three cases:

1. If $f(n) = O(n^c)$ where $c < t$ then $T(n) = O(n^t)$
2. If $f(n) = O(n^c)$ where $c = t$ then $T(n) = O(n^t \log(n))$
3. If $f(n) = O(n^c)$ where $c > t$ then $T(n) = O(n^c)$

Read more at: **Master's Theorem @Wikipedia**

**How does this work?**
Master theorem is mainly derived from recurrence tree method. If we draw recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $O(nc)$ where $c$ is $log_b a$. And the height of recurrence tree is $log_b n$ .

In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then, our result becomes work done at root (Case 3).

Some standard algorithms whose time complexity can be evaluated using Master's Theorem:

1. **Merge Sort:** $T(n) = 2T(n/2) + n$. It falls in case 2 as $c$ is 1 and $t$ [$= log_b a$ ] is also 1. So the solution is $O(n \times \log_2(n))$.
2. **Binary Search:** $T(n) = T(n/2) + 1$. It also falls in case 2 as $c$ is 0 and t [$= log_b a$ ] is also 0. So the solution is $O(\log_2(n))$.
3. **Binary Tree Traversal:** $T(n) = 2T(n/2) + 1$. It falls in case 1 as $c$ is 0 and $t$ [$= log_b a$ ] is 1 [$c < t$]. So the solution is $O(n)$.

Note that it is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master's Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + (n/\log_2(n))$ cannot be solved by using master's theorem.

## Amortized Analysis

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.

The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

Let us consider an example of a simple Hash-Table Insertion. How do we decide Table size? There is a trade-off between space and time, if we make our Hash-Table size big, search time becomes fast, but space required becomes high. But, if we make our Hash-Table size smaller, our search time increases. Therefore, there's always a trade-off between time and space.

**Dynamic Table**
The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full:

1. Allocate memory for a larger table of size, typically twice the old table.
2. Copy the contents of old table to the new table.
3. Free the old table.

If the table has space available, we simply insert new item in available space.
Examples of Dynamic Tables in different languages are: `std::vector` in C++, `ArrayList` in Java, `Lists` in Python, etc.

**What is the time complexity of *n* insertions using the above scheme (dynamic table)?**
If we use asymptotic analysis, the worst case cost of an insertion is O($n$). Therefore, worst case cost of $n$ inserts is $n \times O(n)$ which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for $n$ insertions as all insertions don't take O($n$) time.

After amortized analysis, we can say that the complexity is O($n$), as creating and copying is rare but expensive operation, and hence can be neglected while computing complexity.

## Time Complexity Table

Assume that for a given *problem*, we have several algorithms whose *complexities* (*Big-O*) vary from one another, *i.e.*, we have *solutions* with time complexities as $O(2^n)$, $O(n^3)$, $O(n^2)$, $O(n \times \log_2(n))$, $O(n)$, $O(\log_2(n))$, and $O(1)$, where, for every *solution*, '$n$' is the *input size* to the algorithm.
Now, if we want to find the *actual time taken* (in seconds) by the different solutions, we need to know the following *important parameters*:

1. The *Size of the Input* (What is our '$n$'?).
2. The *Number of Machine Instructions executed internally* by the system, when we execute a *statement(s)* in our implementation of the algorithm [***Machine Instructions***[1] depend on the ***Instruction Set*** of the system that we are using, to implement the algorithm].
3. The *Clock Frequency* of the *Processor* on which the algorithm is implemented.

In the real world, the aforementioned parameters, are *highly system dependent*, therefore, these parameters keep on changing from one system to another system.

So for simplicity, we take all the three parameters mentioned above, as:

1. *Size of the Input(n)*: $n = 10^6$. We took this input size, because it is not too big, and not too small for our analysis.
2. *Number of Machine Instructions executed internally*: For this parameter, we can assume that a single statement like a *loop/iteration*, is internally, just a single machine instruction, *i.e.*, 1 *iteration* = 1 *machine instruction* in our system.
3. *Clock Frequency* = 1GHz, i.e., if a processor's speed is 1GHz, then in a second, the processor can generate $10^9$ *clock cycles*.
    - If we assume that 1 *clock cycle* = 1 *instruction*, then:
        - $10^9$ *clock cycles per second* => $10^9$ *instructions per second*.
        - If our processor can execute $10^9$ instructions per second, then execution time for a single instruction will be, $T = \frac{1}{10^9}$ *seconds* = 1 *ns*

Therefore, for the values, $n = 10^6$ and $T = \frac{1}{10^9}$ *seconds,* our Time Complexity Table is:

| **Big- O** | $n^3$ | $n^2$ | $n \times \log_2(n)$ | $n$ | $\log_2(n)$ | 1 | $2^n$ [$n$=30] | $2^n$ [$n$=60] |
|---|---|---|---|---|---|---|---|---|
| **Total #Instructions*** | $10^{18}$ | $10^{12}$ | $2 \times 10^7$ | $10^6$ | 20 | 1 | $2^{30} \approx 10^9$ | $2^{60} \approx 10^{18}$ |
| **Time Taken**** | 31.7 *yrs* | 16 *mins* | 0.02 *sec* | 1 ms | 20 *ns* | 1 *ns* | 1 *sec* | 31.7 *yrs* |

**Total #Instructions:* We have $n = 10^6$, therefore, $O(n^3) = (10^6)^3 = 10^{18}$ instructions to be executed.
***Time Taken:* This is the total time taken by the processor to execute the given solution, *i.e.*, if a solution for some problem takes $O(n^3)$ time, then, for $n = 10^6$, we have $10^{18}$ instructions to be executed by the processor, as shown above in the green column. Therefore, in a 1GHz processor ($T = \frac{1}{10^9}$ *seconds)*: [Note: T is the execution time for a single instruction]

1 *instruction takes* $\frac{1}{10^9}$ *seconds*, then,

$10^{18}$ *instruction will take* $\frac{10^{18}}{10^9}$ *seconds* $= 10^9$ *seconds* $= \frac{10^9}{60 \times 60 \times 24 \times 365}$ *years* $\approx 31.7$ *years*.

[1] **More on Machine Instructions** (must watch).

## Space Complexity

The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity:

*Auxiliary Space*: The extra/temporary space used by an algorithm.

*Space Complexity*: The total space taken by the algorithm with respect to its input size. Space complexity includes both *auxiliary space* and *space used by input*.

For example, if we want to compare standard sorting algorithms on the basis of space, then *auxiliary space* would be better a criteria for *space complexity*. Merge Sort uses $O(n)$ auxiliary space, Insertion Sort and Heap Sort use O(1) *auxiliary space*. Space complexity of all these sorting algorithms is $O(n)$ though.

## Problems & Solutions

1. What is the time complexity of following function `fun()`?

```
void fun() {
    int i, j;
    for (i = 1; i <= n; i++)
        for (j = 1; j <= log2(i); j++)
            printf("hello!!");
}
```

**Solution:**

a. Time Complexity of the above function can be written as :
$O(\log_2(1)) + O(\log_2(2)) + O(\log_2(3)) + \ldots + O(\log_2(n))$ which is $O(\log_2(n!))$

b. Order of growth of $\log_2(n!)$ and $n \times \log_2(n)$ is same for large values of $n$, *i.e.*, $O(\log_2(n!)) = O(n \times \log_2(n))$. So time complexity of fun() is $O(n \times \log_2(n))$.

c. The expression $O(\log_2(n!)) = O(n \times \log_2(n))$ can be easily derived from Stirling's approximation (or Stirling's formula) *i.e.* :
$\log_2(n!) = (n \times \log_2(n)) - n + O(\log_2(n))$

2. What is the time complexity of below function?

```
void fun(int n, int k) {
    for (int i = 1; i <= n; i++) {
        int p = pow(i, k);
        for (int j = 1; j <= p; j++)
            // Some O(1) work
    }
}
```

**Solution:** Time complexity of above function can be written as $1^k + 2^k + 3^k + \ldots n^k$.
Let us try few examples:

a. $k = 1$: Sum $= 1 + 2 + 3 + \ldots + n$
$= (n \times (n + 1))/2$
$= (n^2) + (n/2)$

b. $k = 2$: Sum $= 1^2 + 2^2 + 3^2 + \ldots + n^2$
$= (n \times (n + 1) \times (2n + 1))/6$
$= (n^3/3) + (n^2/2) + (n/6)$

c. $k = 3$: Sum $= 1^3 + 2^3 + 3^3 + \ldots + n^3$
$= (n^2 \times (n + 1)^2)/4$
$= (n^4/4) + (n^3/2) + (n^2/4)$

In general, asymptotic value can be written as $((n^{k+1})/(k + 1))$, *i.e.*, $O((n^{k+1})/(k + 1))$

3. What is the time complexity of the following code:

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++ ) {
        for (int k = i + 1; k <= n; k++) {
            ans++;
        }
    }
}
```

**Solution:** $T(n) = \sum_{k=1}^{n} k(n-k) = O(n^3)$